Insights of Particle Image Velocimetry

JF Krawczynski

2025-10-03

Table of contents

About

This site hosts the course materials for fluid mechanics measurements by PIV for the course unit **UM5MEE12 - Experimental Methods and Data Processing** taught in the Master of Engineering Sciences at the Faculty of Engineering of Sorbonne University. The aim is to understand the basic concepts inherent to measuring velocity fields in fluid flows.

> **Note**
>
> This course is partly adapted from the documentation of OpenPIV, licensed under the GNU General Public License v2.0.

The course content is divided into two parts:

- The theoretical aspects of the course, including:
  - the fundamental physical considerations related to the tracer particles used to measure the flow
  - the algorithms developed to obtain the velocity vector field from the analysis of particle images
- The practical aspects:
  - a numerical part to better understand the evaluation methods using PIV. The images processed will be generated synthetically
  - an experimental part to understand the trade-offs required to obtain high-quality images for processing

In this course, we will mainly use the following tools, with which students should be familiar:

- Python
- Numpy
- Matplotlib

> **Tip**
>
> A PDF version of this website is available for offline viewing (please avoid printing!).

Assessment

Assessment will be based on a short lab report (for the numerical part) and a project report focusing on the analysis of experimental data obtained during the lab sessions.

Schedule

Here is the planned organization of the sessions:

#	Date	Topic
1	09/24/2025	Introduction to velocity field measurement by PIV, Lecture 1
2	10/01/2025	Algorithms and experimental specifics, Lab 1 + Lab 2
3	10/08/2025	Algorithms and experimental specifics, Lab 1 + Lab 2

Recommended Readings and Media

Below is a list of useful resources for your project, organized by topic.

Python for Scientific Programming

- Un Zeste de Python
- Les bases de numpy et matplotlib
- From Python to Numpy
- Scientific Visualization: Python + Matplotlib (N. P. Rougier 2021)

Scientific outputs: figures and reports

- Ten Simple Rules for Better Figures (M. D. Rougier Nicolas P. 2014)
- Wikibook LaTeX
- Typst

Theory

Basics of PIV

Using simple mathematical knowledge and existing algorithms written with Python, Numpy, Scipy, we will introduce the basics of PIV.

All the images used can be found in the images directory of the repository.

What is Particle Image Velocimetry (PIV) about?

"Particle" Image Velocimetry (PIV) is a non-intrusive state-of-the-art technique (Adrian 1991), (Raffel 2007) to get the velocity field of the flow being studied from images of small particles, called tracers.

It is based on image recording of the illuminated flow field using seeding particles (called tracers) which, when sufficiently small compared to the smallest characteristic length scales of the flow, are assumed to faithfully follow the flow dynamics (the degree to which the particles faithfully follow the flow is represented by the Stokes number). In practice, common sizes of the tracer particles are in the order of 5-100 microns. The entrained particles are generally made visible in a cross-section of the flow being studied by forming a coherent light sheet. In practice, the flow is illuminated twice using a laser light sheet, forming a plane where a camera is focused. The time delay between the pulses depends on the mean velocity and the image magnification. It is assumed that the tracer particles follow the local flow velocity between the two consecutive illuminations. The light scattered from the tracer particles is then imaged via an optical lens on a digital camera. The displacement of the particle images between two consecutive light pulses (respectively, frames) is determined through evaluation of the spatial cross-correlation function and image processing tools as implemented in OpenPIV.

The effectiveness of the measurement results strongly depends on a large number of parameters such as particles concentration, size distribution and shape, illumination source, recording device, and synchronization between the illumination, acquisition and recording systems (Huang 1997). An appropriate choice of the different parameters of the cross correlation analysis (e.g., interrogation area, time between pulses, scaling) will influence the results accuracy.

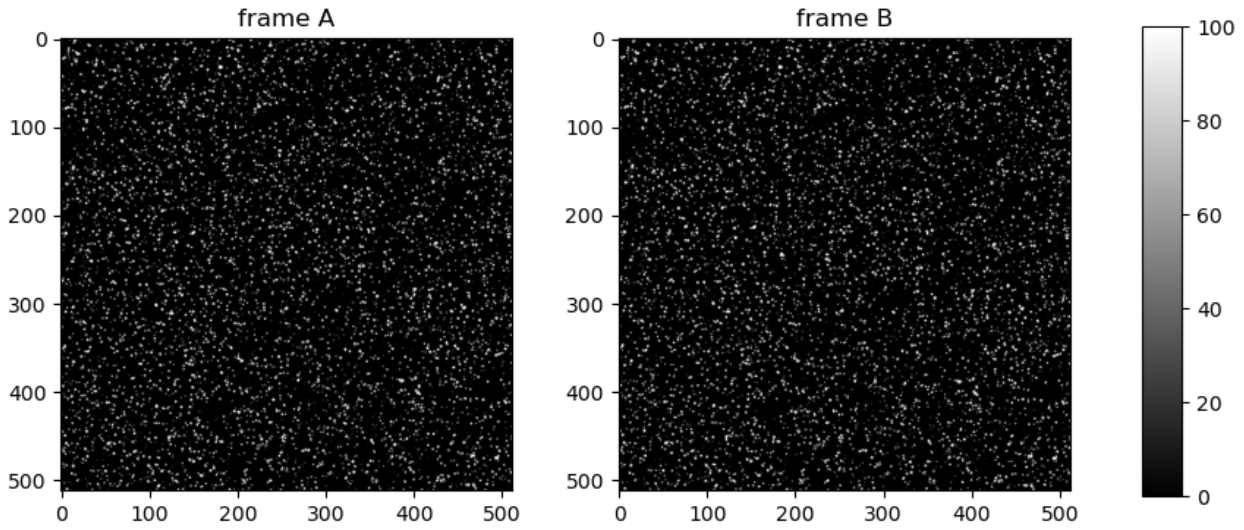
How to estimate a velocity-field from a couple of grayscale particle images?

This tutorial will follow the simplest analysis path from the two images to the velocity field and some post-analysis.

```
# import the standard numerical and plotting packages
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
# import what is necessary from OpenPIV
from openpiv import tools, pyprocess, validation, filters, scaling
```

We have downloaded some sample images from a standard PIV images project, see pivchallenge

```
# load the images
a = tools.imread("./images/B005_1.tif")
b = tools.imread("./images/B005_2.tif")

fig, axs = plt.subplots(1, 2, figsize=(9, 4))

img = axs[0].imshow(a, vmin=0, vmax=100, cmap=plt.cm.gray)
axs[0].set_title('frame A')

img = axs[1].imshow(b, vmin=0, vmax=100, cmap=plt.cm.gray)
axs[1].set_title('frame B')

cbar = fig.add_axes([0.95, 0.1, 0.03, 0.8])
fig.colorbar(img, cax=cbar)
plt.show()
```

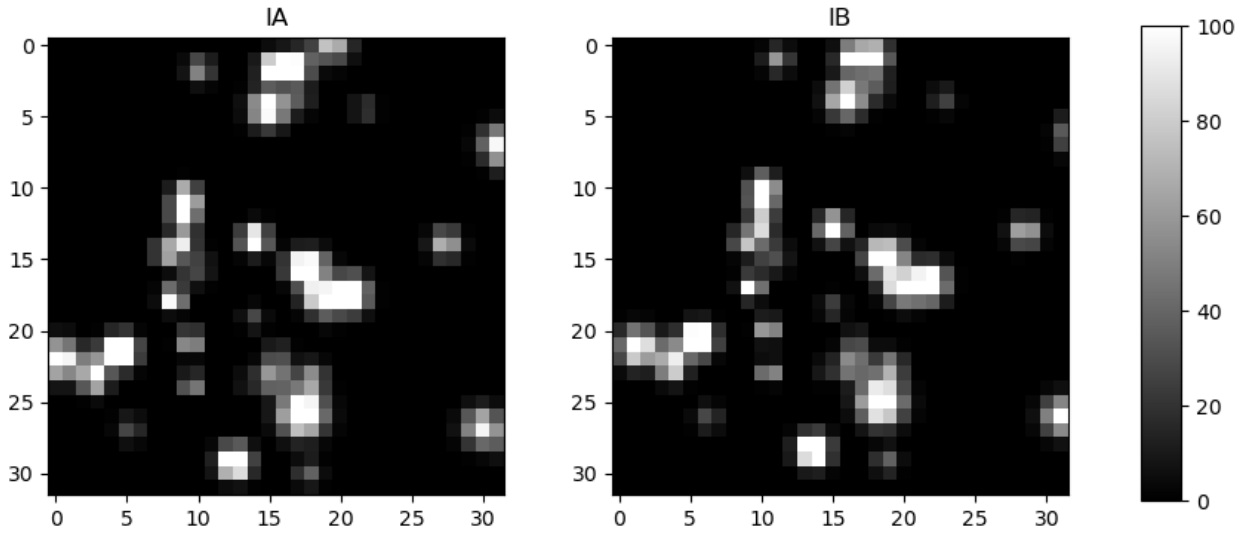


The two images show the particles at two different times. But these images are way to big and contain too many particles to manually track the movement of individual particles from one frame to the other. Instead, we can analyze small regions of interest, called interrogation windows (IW).

Typically, we can start with a size of 32 x 32 pixels or smaller. Until recently, the fast algorithms used powers of 2, so the historical sizes are always powers of 2: 8, 16, 32, 64, 128, …

Let's take the first top left windows from each image.

```
win_size = 32

a_win = a[:win_size, :win_size].copy()
b_win = b[:win_size, :win_size].copy()

fig, axs = plt.subplots(1, 2, figsize=(9, 4))
img = axs[0].imshow(a_win, vmin=0, vmax=100, cmap=plt.cm.gray)
axs[0].set_title('IA')

img = axs[1].imshow(b_win, vmin=0, vmax=100, cmap=plt.cm.gray)
axs[1].set_title('IB')

cbar = fig.add_axes([0.95, 0.1, 0.03, 0.8])
fig.colorbar(img, cax=cbar)
plt.show()
```



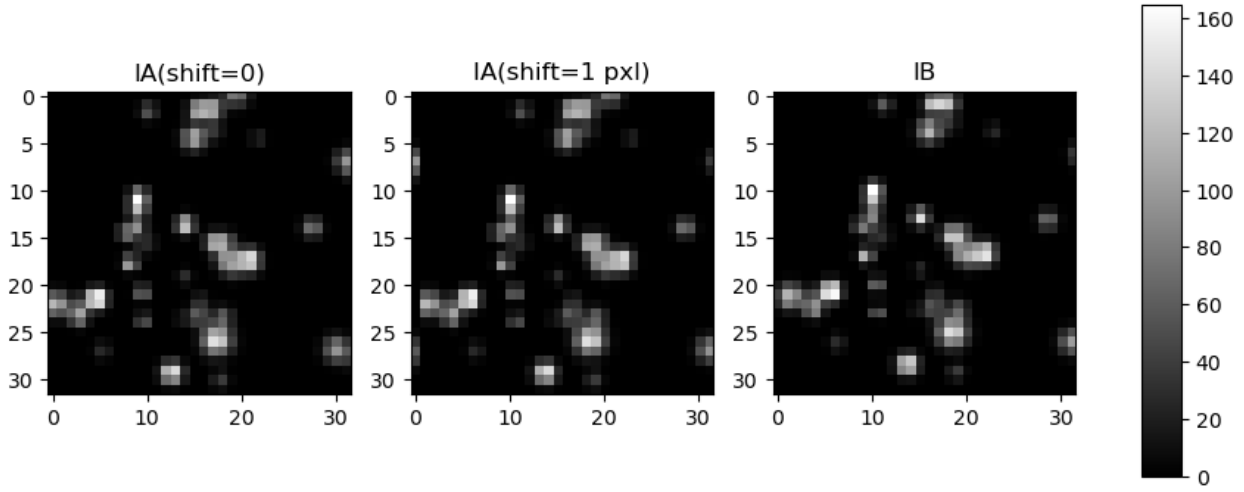
If you tried to manually track the movement of individual particles from frame A to frame B, it would be extremely time-consuming, especially for large image regions containing many particles. Manually identifying and matching every single particle would quickly become tedious and impractical for even a coarse velocity field estimation. That's why automated methods, such as using correlation techniques or least squares approaches, are preferred in Particle Image Velocimetry (PIV). These methods can efficiently analyze the displacement of all particles within interrogation windows and generate a velocity field much faster and more accurately than manual matching.

We can find out the distance that all the particles moved between frame A and frame B using the principles of least squares or correlations, but let's first try to get it manually.

If we try to shift the window IA by some pixels along the horizontal and/or vertical directions, we shall see an image that gets closer in resemblance to the window IB.

Assignment: Modify the code below to minimize the difference between the shifted IA and IB.

```
fig, axs = plt.subplots(1, 3, figsize=(9, 4))

img = axs[0].imshow(a_win, cmap=plt.cm.gray)
axs[0].set_title("IA(shift=0)")

img = axs[1].imshow(np.roll(a_win, (0, 1), axis=(0, 1)), cmap=plt.cm.gray)
axs[1].set_title("IA(shift=1 pxl)")

img = axs[2].imshow(b_win, cmap=plt.cm.gray)
axs[2].set_title("IB")

cbar = fig.add_axes([0.95, 0.1, 0.03, 0.8])
fig.colorbar(img, cax=cbar)
plt.show()
```

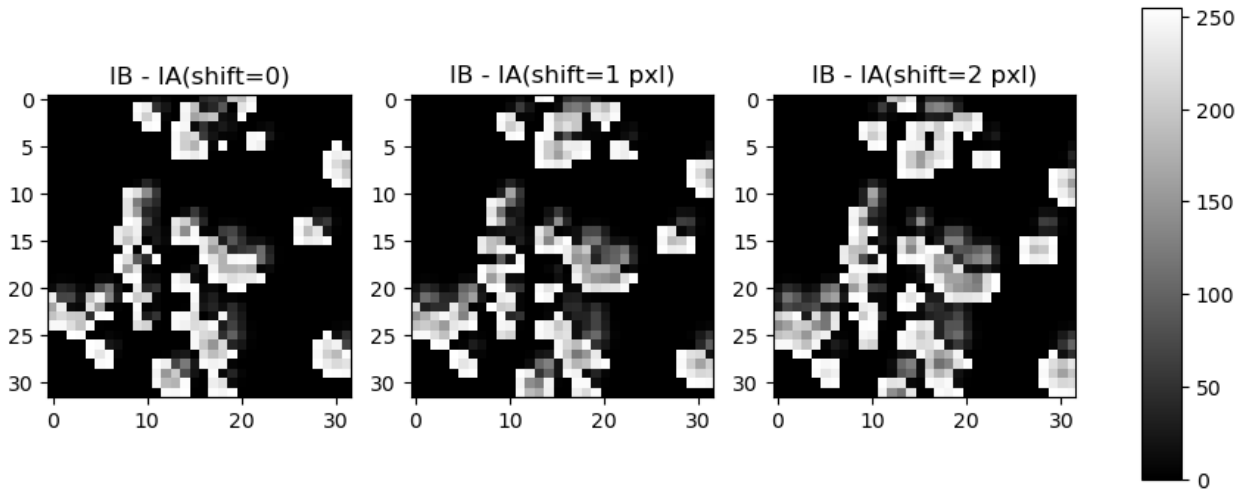


If we now subtract from IB the shifted IA, we shall see how good the shift predicts the real displacement between the two.

Assignment: Modify the code below to minimize the difference between the shifted IA and IB. Share your comments.

```
fig, axs = plt.subplots(1, 3, figsize=(9, 4))

img = axs[0].imshow(b_win - a_win, cmap=plt.cm.gray)
axs[0].set_title("IB - IA(shift=0)")
```

```
img = axs[1].imshow(b_win - np.roll(a_win, (1, 0), axis=(0, 1)), cmap=plt.cm.gray)
axs[1].set_title("IB - IA(shift=1 pxl)")

img = axs[2].imshow(b_win - np.roll(a_win, (2, 0), axis=(0, 1)), cmap=plt.cm.gray)
axs[2].set_title("IB - IA(shift=2 pxl)")

cbar = fig.add_axes([0.95, 0.1, 0.03, 0.8])
fig.colorbar(img, cax=cbar)
plt.show()
```



Let's try to find the best shift algorithmically: shift and calculate the sum of squared differences, the minimum of which should be the best shift.

```
def match_template(img, template, maxroll=8):
    # img: input image to compare
    # template: image to match against
    # maxroll: max nbr of pxl to shift in each dir. Default value is 8.
    best_dist = np.inf
    best_shift = (-1, -1)
    for y in range(-maxroll,maxroll):
        for x in range(-maxroll,maxroll):
            # calculate Euclidean distance
            dist = np.sqrt(np.sum((img - np.roll(template, (y, x), axis=(0, 1))) ** 2))
            if dist < best_dist:
                best_dist = dist
                best_shift = (y, x)
    return (best_dist, best_shift)
```

Let's check that it works by manually shifting the same image (IA):

```
match_template(np.roll(a_win, (2, 0), axis=(0, 1)), a_win)
```

```
(0.0, (2, 0))
```

Indeed, when we find the correct shift, we get zero distance. It's not so in real images:

```
match_template(b_win, a_win)
```

```
(123.80226169178009, (-1, 1))
```

Well, this is not the true displacement, but it gives a hint.

We could draw this as a vector of velocity

$$u = \frac{\Delta x \text{ pixels}}{\Delta t}, \qquad v = \frac{\Delta y \text{ pixels}}{\Delta t}$$

where Δt is the time interval (delay) between the two images (or two laser pulses).

The problem is that shifting each image and repeating the loop many times is impractical.

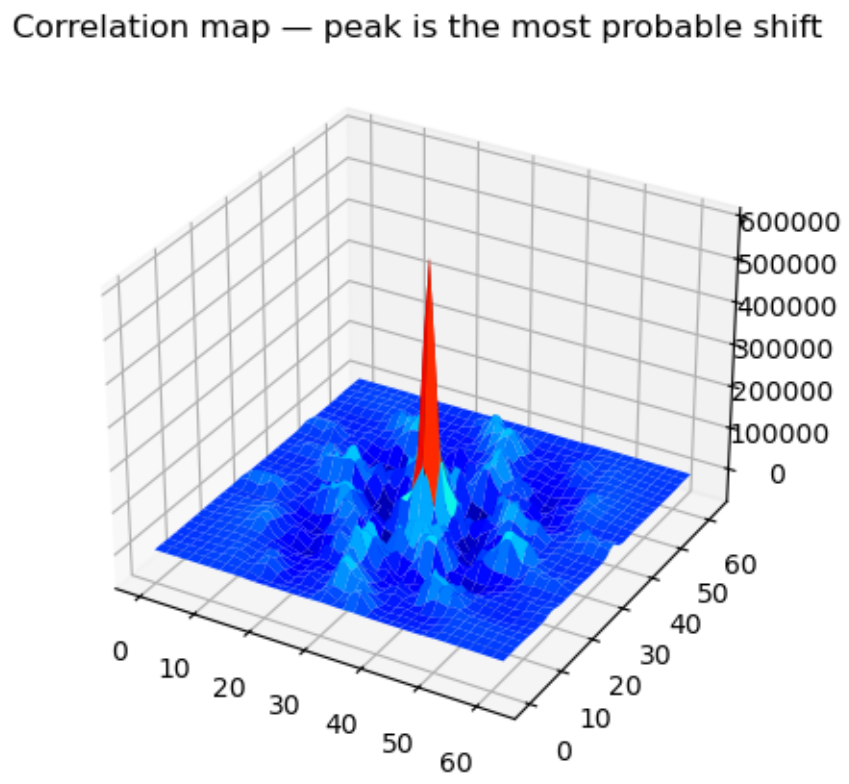
However, one can get it by using a different matching principle, based on the property called cross-correlation (cross because we use two different images). This is an efficient computational algorithm to find out the right shift. You can see more details here: http://paulbourke.net/miscellaneous/correlate/.

```
from scipy.signal import correlate

cross_corr = correlate(b_win - b_win.mean(), a_win - a_win.mean(), method="fft")
# Note that it's approximately twice as large as the original windows, as we
# can shift a_win by a maximum of its size horizontally and vertically
# while still maintaining some overlap between the two windows.
print("Size of the correlation map: %d x %d" % cross_corr.shape)
```

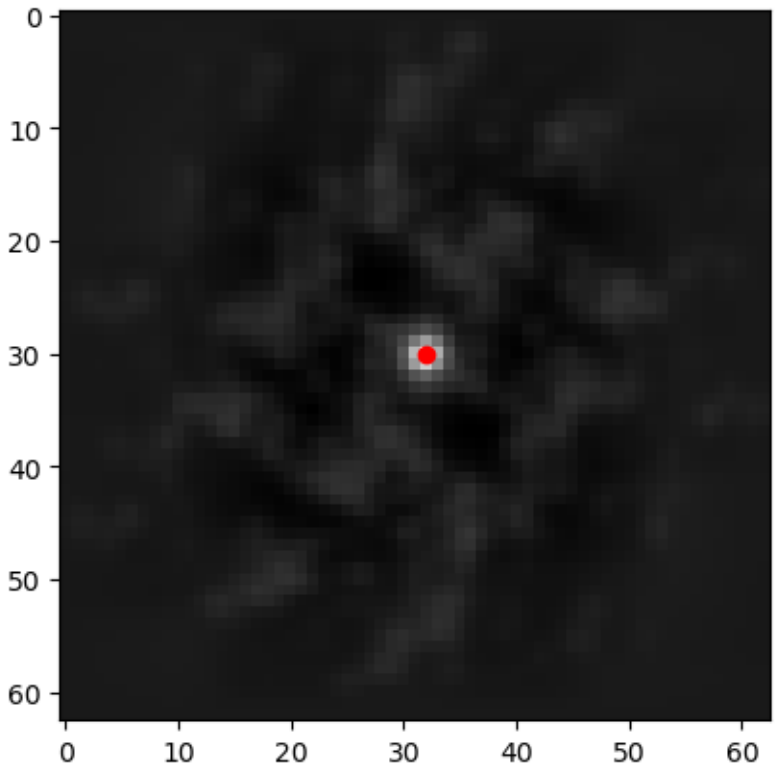
```
Size of the correlation map: 63 x 63
```

```
# let's see what the cross-correlation looks like
#from mpl_toolkits.mplot3d import Axes3D
Y, X = np.meshgrid(np.arange(cross_corr.shape[0]), np.arange(cross_corr.shape[1]))

fig = plt.figure()
ax = fig.add_subplot(projection="3d")
ax.plot_surface(Y, X, cross_corr, cmap='jet', linewidth=0.2)  # type: ignore
plt.title("Correlation map - peak is the most probable shift")
plt.show()
```

```
# let's see the same correlation map, from above
plt.imshow(cross_corr, cmap=plt.cm.gray)
y, x = np.unravel_index(cross_corr.argmax(), cross_corr.shape)
print(f"{y=}, {x=}")
plt.plot(x, y, "ro")
plt.show()
```




```
y=30, x=32
```



The image of the correlation map shows the same result that we got manually looping. We need to substract the center of symmetry `(31, 31)` to get the estimated displacement.

```
dy, dx = y - 31, x - 31
print(f"{dy=}, {dx=}")
```

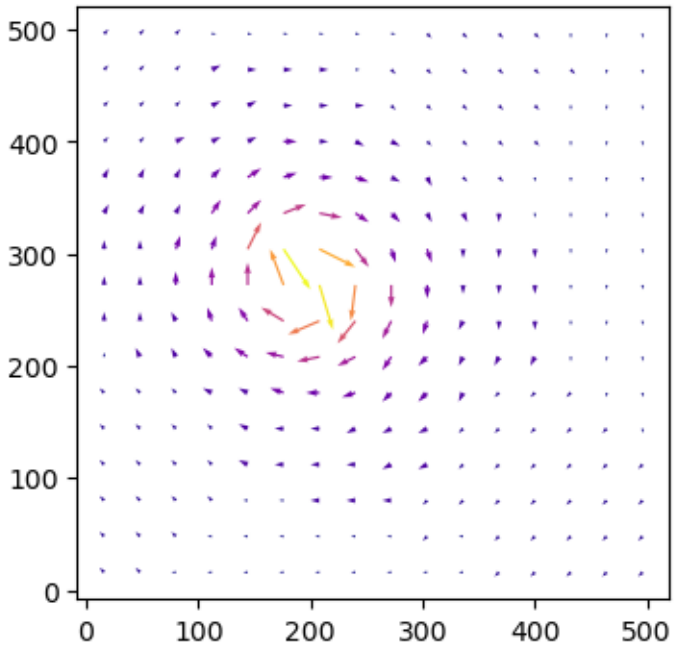
```
dy=-1, dx=1
```

We can get the first velocity field by repeating this analysis for all small windows. Let's take 32 x 32 pixels windows from each image and do the loop:

```
def vel_field(curr_frame, next_frame, win_size):
    ys = np.arange(0, curr_frame.shape[0], win_size)
    xs = np.arange(0, curr_frame.shape[1], win_size)
    dys = np.zeros((len(ys), len(xs)))
    dxs = np.zeros((len(ys), len(xs)))
    for iy, y in enumerate(ys):
        for ix, x in enumerate(xs):
            int_win = curr_frame[y : y + win_size, x : x + win_size]
            search_win = next_frame[y : y + win_size, x : x + win_size]
            cross_corr = correlate(
                search_win - search_win.mean(), int_win - int_win.mean(), method="fft"
```

```
            )
            dys[iy, ix], dxs[iy, ix] = (
                np.unravel_index(np.argmax(cross_corr), cross_corr.shape)
                - np.array([win_size, win_size])
                + 1
            )
    # draw velocity vectors from the center of each window
    ys = ys + win_size / 2
    xs = xs + win_size / 2
    return xs, ys, dxs, dys
```

```
xs, ys, dxs, dys = vel_field(a, b, 32)
norm_drs = np.sqrt(dxs ** 2 + dys ** 2)

fig, ax = plt.subplots(figsize=(9, 4))
# we need these flips on y since quiver uses a bottom-left origin, while our
# arrays use a top-right origin
ax.quiver(
    xs,
    ys[::-1],
    dxs,
    -dys,
    norm_drs,
    cmap="plasma",
    angles="xy",
    scale_units="xy",
    scale=0.25,
)
ax.set_aspect("equal")
plt.show()
```



If you've followed along this far, great! Now you understand the basics.

We can also try out a variant of this that uses a search window larger than the interrogation window instead of relying on zero-padding. By avoiding using zero-padding around the search window, movement detection should theoretically be a bit better, assuming that the window sizes are chosen well.

```
def vel_field_asymmetric_wins(
    curr_frame, next_frame, half_int_win_size, half_search_win_size
):
    ys = np.arange(half_int_win_size[0], curr_frame.shape[0], 2 * half_int_win_size[0])
    xs = np.arange(half_int_win_size[1], curr_frame.shape[1], 2 * half_int_win_size[1])
    dys = np.zeros((len(ys), len(xs)))
    dxs = np.zeros((len(ys), len(xs)))
    for iy, y in enumerate(ys):
        for ix, x in enumerate(xs):
            int_win = curr_frame[
                y - half_int_win_size[0] : y + half_int_win_size[0],
                x - half_int_win_size[1] : x + half_int_win_size[1],
            ]
            search_win_y_min = y - half_search_win_size[0]
            search_win_y_max = y + half_search_win_size[0]
            search_win_x_min = x - half_search_win_size[1]
            search_win_x_max = x + half_search_win_size[1]
            truncated_search_win = next_frame[
                max(0, search_win_y_min) : min(b.shape[0], search_win_y_max),
                max(0, search_win_x_min) : min(b.shape[1], search_win_x_max),
```

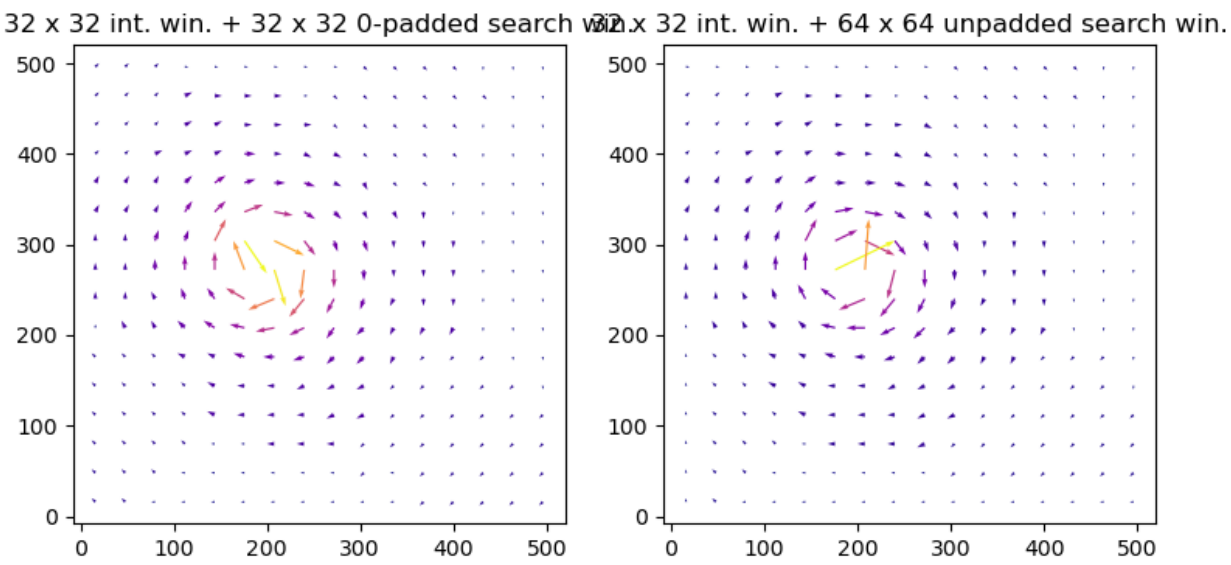
```
            ]
            cross_corr = correlate(
                truncated_search_win - np.mean(truncated_search_win),
                int_win - np.mean(int_win),
                mode="valid",
                method="fft",
            )
            dy, dx = np.unravel_index(np.argmax(cross_corr), cross_corr.shape)
            # if the top of the search window got truncated, shift the origin
            # up to the top edge of the (non-truncated) search window
            if search_win_y_min < 0:
                dy += -search_win_y_min
            # if the left of the search window got truncated, shift the origin
            # over to the left edge of the (non-truncated) search window
            if search_win_x_min < 0:
                dx += -search_win_x_min
            # shift origin to the center of the search window
            dy -= half_search_win_size[0] - half_int_win_size[0]
            dx -= half_search_win_size[1] - half_int_win_size[1]
            dys[iy, ix] = dy
            dxs[iy, ix] = dx
    return xs, ys, dxs, dys
```

```
int_win_size = np.array([32, 32])
print(f"{int_win_size=}")
assert np.all(np.array(a.shape) % int_win_size == 0)
assert np.all(int_win_size % 2 == 0)
half_int_win_size = int_win_size // 2

search_win_size = int_win_size * 2
print(f"{search_win_size=}")
assert np.all(search_win_size % 2 == 0)
half_search_win_size = search_win_size // 2
assert np.all(search_win_size > int_win_size)
print(
    "max velocity that can be detected with these window sizes: "
    + f"{half_search_win_size - half_int_win_size}"
)
```

```
int_win_size=array([32, 32])
search_win_size=array([64, 64])
max velocity that can be detected with these window sizes: [16 16]
```

Making the search window larger compared to the interrogation window would allow for larger velocities to be detected.

```
xs_asym, ys_asym, dxs_asym, dys_asym = vel_field_asymmetric_wins(
    a, b, half_int_win_size, half_search_win_size
)
norm_drs_asym = np.sqrt(dxs_asym ** 2 + dys_asym ** 2)

fig, axs = plt.subplots(1, 2, figsize=(9, 4))
axs[0].quiver(
    xs,
    ys[::-1],
    dxs,
    -dys,
    norm_drs,
    cmap="plasma",
    angles="xy",
    scale_units="xy",
    scale=0.25,
)
axs[1].quiver(
    xs_asym,
    ys_asym[::-1],
    dxs_asym,
    -dys_asym,
    norm_drs_asym,
    cmap="plasma",
    angles="xy",
    scale_units="xy",
    scale=0.25,
)
axs[0].set_title(
    f"{win_size} x {win_size} int. win. + "
    f"{win_size} x {win_size} 0-padded search win."
)
axs[1].set_title(
    f"{int_win_size[0]} x {int_win_size[1]} int. win. + "
    f"{search_win_size[0]} x {search_win_size[0]} unpadded search win."
)
ax.set_aspect("equal")
plt.show()
```


32 x 32 int. win. + 32 x 32 0-padded search win.
32 x 32 int. win. + 64 x 64 unpadded search win.
0
100
200
300
400
500

OpenPIV first tutorial

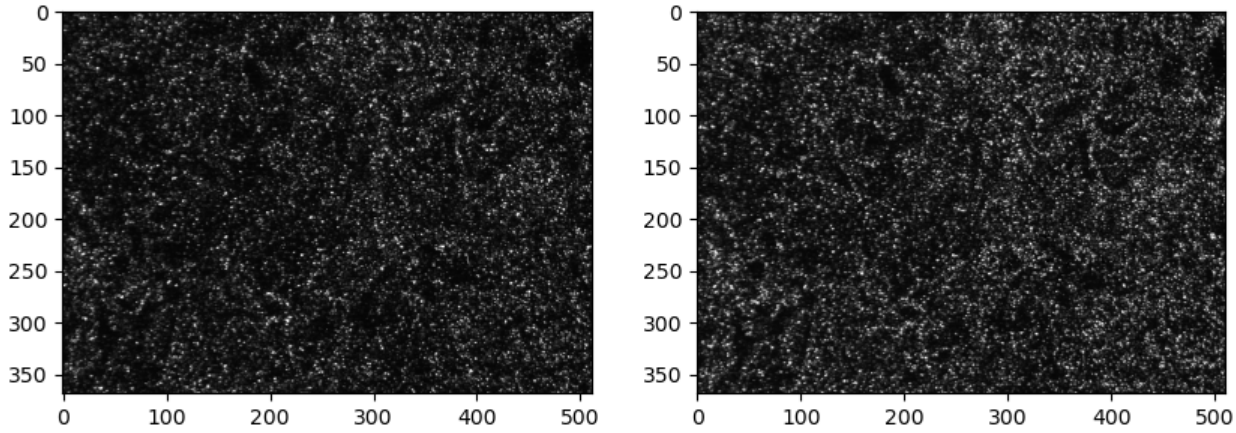
Using open source software, OpenPIV (http://www.openpiv.net), written with Python, Numpy, Scipy (http://www.scipy.org), we will introduce the basics of PIV.

This tutorial will follow the simplest analysis path from the two images to the velocity field and some post-analysis.

```
# import the standard numerical and plotting packages
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
# import what is necessary from OpenPIV
from openpiv import tools, pyprocess, validation, filters, scaling
```

```
# read a pair of PIV images
frame_a  = tools.imread( './images/exp1_001_b.bmp' )
frame_b  = tools.imread( './images/exp1_001_c.bmp' )
```

```
# let's visualize them using matplotlib
fig,ax = plt.subplots(1,2,figsize=(10,8))
ax[0].imshow(frame_a,cmap=plt.cm.gray)
ax[1].imshow(frame_b,cmap=plt.cm.gray)
plt.show()
```



Processing

We are going to use the `extended_search_area_piv function`, wich is a standard PIV cross-correlation algorithm.

This function allows the search area (`search_area_size`) in the second frame to be larger than the interrogation window in the first frame (`window_size`). Also, the search areas can overlap (`overlap`).

The `extended_search_area_piv` function will return three arrays: 1. The `u` component of the velocity field 2. The `v` component of the velocity field 3. The signal-to-noise ratio (`sig2noise`) of the cross-correlation map of each vector. The higher the signal-to-noise ratio, the higher the probability that its magnitude and direction are correct.

```
# define the PIV analysis parameters
winsize = 24 # size of the interrogation window in frame A, in pixels
searchsize = 32  # size of the window in frame B (searchsize is larger or equal to winsize), in
overlap = 12 # overlap between the neighbouring windows, in pixels
dt = 0.02 # time interval of the PIV recording, in sec
```

```
u, v, sig2noise = pyprocess.extended_search_area_piv(
    frame_a.astype(np.int32),
    frame_b.astype(np.int32),
    window_size=winsize,
    overlap=overlap,
    dt=dt,
    search_area_size=searchsize,
    sig2noise_method='peak2peak',
)
```

The function `get_coordinates` finds the center of each interrogation window. This will be useful later on when plotting the vector field.

```
# get a list of coordinates for the vector field
x, y = pyprocess.get_coordinates(
    image_size=frame_a.shape,
    search_area_size=searchsize,
    overlap=overlap,
)
```

Post-processing

Strictly speaking, we are ready to plot the vector field. However, some spurious vectors might locally impact the quality of the results. It might therefore be useful to apply some filtering.

To start, let's use the function `sig2noise_val` to get a mask indicating which vectors have a minimum amount of signal-to-noise ratio. Vectors below a certain threshold are substituted by `NaN`. If you are not sure about which threshold value to use, try taking a look at the signal-to-noise ratio histogram with: `plt.hist(sig2noise.flatten())`.

```
# clean the peaks that are below a quality threshold
invalid_mask = validation.sig2noise_val(
    sig2noise,
    threshold = 1.3,
)
```

Another useful function is `replace_outliers`, which will find outlier vectors and substitute them with an average of neighboring vectors. The larger the `kernel_size`, the larger the considered neighborhood. This function uses an iterative image inpainting algorithm. The number of iterations can be chosen via `max_iter`.

```
# replace those that are masked as bad vectors with local interpolation
u, v = filters.replace_outliers(
    u, v,
    invalid_mask,
    method='localmean',
    max_iter=3,
    kernel_size=3,
)
```

Next, we are going to convert pixels to millimeters.

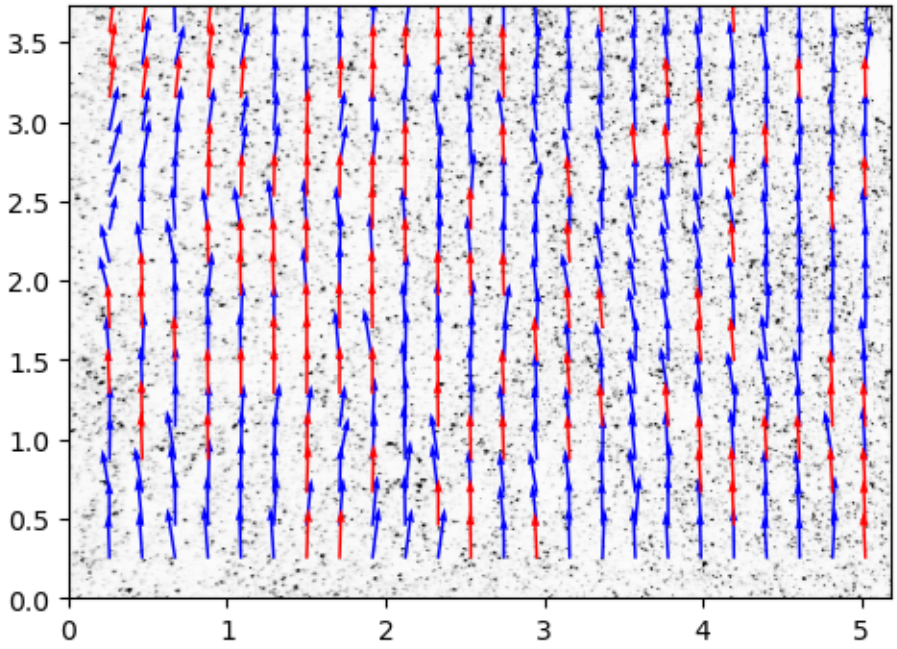
```
# scale the results from pix/dt to mm/sec
x, y, u, v = scaling.uniform(x, y, u, v, scaling_factor = 96.52 ) # 96.52 pixels/millimeter
```

Plot and save the results

The vector field can be plotted with `display_vector_field`. Vectors with signal-to-noise ratio below the threshold are displayed in red.

```
import pathlib

fig, ax = plt.subplots(figsize=(9,4))
tools.display_vector_field(
    pathlib.Path('exp1_001.txt'),
    ax=ax, scaling_factor=96.52,
    scale=50, # scale defines here the arrow length
    width=0.0035, # width is the thickness of the arrow
```

```
    on_img=True, # overlay on the image
    image_name= str('./images/exp1_001_b.bmp'),
)
plt.show()
```



The function `save` is used to save the vector field to a ASCII tabular file.

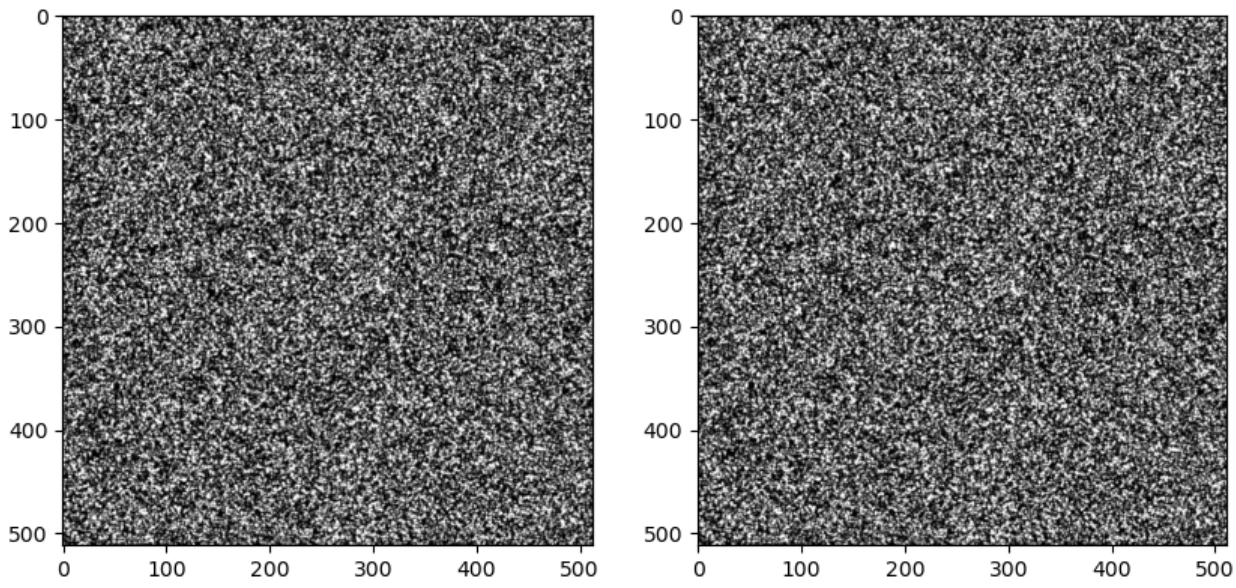
```
tools.save('exp1_001.txt' , x, y, u, v, invalid_mask)
```

Another example

Use any pair of images that you can access via URL

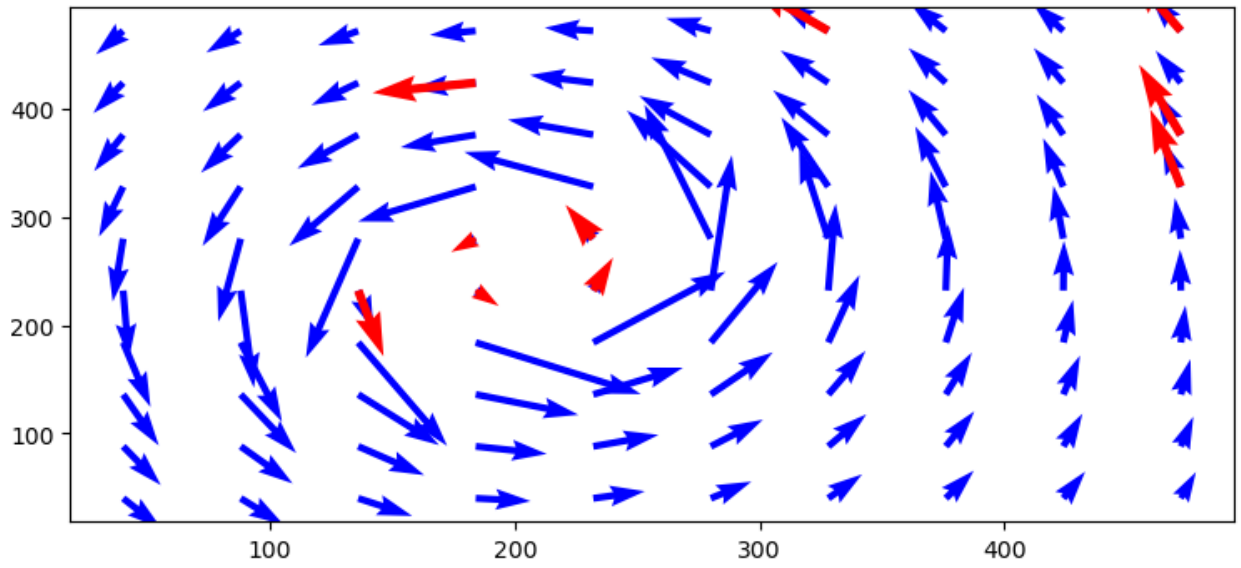
For instance we can use images from PIV Challenge http://www.pivchallenge.org/

```
frame_a = tools.imread('http://www.pivchallenge.org/pub/B/B001_1.tif')
frame_b = tools.imread('http://www.pivchallenge.org/pub/B/B001_2.tif')
#frame_a = tools.imread("./images/B005_1.tif")
#frame_b = tools.imread("./images/B005_2.tif")
fig,ax = plt.subplots(1,2,figsize=(10,8))
ax[0].imshow(frame_a,cmap=plt.cm.gray)
ax[1].imshow(frame_b,cmap=plt.cm.gray)
plt.show()
```




```
winsize = 32 # pixels
searchsize = 64  # pixels, search in image B
overlap = 16 # pixels
dt = 1.0 # sec
u, v, sig2noise = pyprocess.extended_search_area_piv( frame_a.astype(np.int32), frame_b.astype(
#x, y = pyprocess.get_coordinates( image_size=frame_a.shape, window_size=winsize, overlap=overl
```

```
x, y = pyprocess.get_coordinates(
    image_size=frame_a.shape,
    search_area_size=searchsize,
    overlap=overlap,
)
#u, v, mask = validation.sig2noise_val( u0, v0, sig2noise, threshold = 1.1 )
#u, v = filters.replace_outliers( u, v, method='localmean', max_iter=10, kernel_size=2)
#Post-processing
invalid_mask = validation.sig2noise_val(
    sig2noise,
    threshold = 1.1,
)

u, v = filters.replace_outliers(
    u, v,
    invalid_mask,
    method='localmean',
    max_iter=10,
    kernel_size=2,
)
# x, y, u, v = scaling.uniform(x, y, u, v, scaling_factor = 96.52 )

plt.figure(figsize=(9,4))
plt.quiver(x,y,u,v,color='b')
plt.quiver(x[invalid_mask],y[invalid_mask],u[invalid_mask],v[invalid_mask],color='r')
plt.show()
```



Advanced PIV techniques

This page provides an in-depth overview of advanced Particle Image Velocimetry (PIV) techniques implemented in the OpenPIV Python package. It covers window deformation methods, multi-pass processing, vector validation, and image preprocessing techniques that go beyond basic PIV analysis.

How window deformation works

Window deformation is an advanced technique that improves PIV accuracy by deforming interrogation windows based on previous displacement estimates. This is particularly useful for flows with high shear or rotation.

Window deformation iteratively deforms the interrogation windows to account for flow gradients within each window. This helps overcome the limitations of standard PIV which assumes uniform displacement within each interrogation region.

OpenPIV supports two window deformation methods. 1. Symmetric Deformation: Both images are deformed by half the displacement in opposite directions. This is more accurate but computationally intensive. 2. Second Image Deformation: Only the second image is deformed. This is faster but potentially less accurate.

How multi-pass processing works

Multi-pass processing iteratively refines PIV analysis by using results from previous passes to guide subsequent analysis with smaller interrogation windows. This approach significantly improves spatial resolution and accuracy. 1. First pass uses large interrogation windows for robustness 2. Subsequent passes use smaller windows for better spatial resolution 3. Each pass uses results from the previous pass to deform windows

Vector Validation Techniques

Vector validation is crucial for identifying and removing spurious vectors from PIV results. OpenPIV implements several validation methods that can be used alone or in combination:

1. Global Validation (`global_val`): Rejects vectors outside specified displacement ranges.

2. Global Statistical Validation (`global_std`): Rejects vectors that deviate from the mean by more than a specified number of standard deviations.
3. Signal-to-noise Validation (`sig2noise_val`): Rejects vectors with a low signal-to-noise ratio.
4. Local Median Validation (`local_median_val`): Rejects vectors that deviate significantly from their local neighborhood median.
5. Normalized Local Median Validation (`local_norm_median_val`): A normalized version of local median validation that accounts for local flow variations.

Vector Replacement

After validation, identified outliers can be replaced using different methods:

1. `localmean`: Replaces outliers with the local neighborhood mean
2. `disk`: Uses a disk-shaped kernel for replacement
3. `distance`: Weighted average based on distance

Configuring Advanced PIV Analysis

Advanced PIV techniques are configured using the `PIVSettings` class, which provides a centralized way to control all aspects of the analysis.

Key Settings

- `windowsizes`: Tuple of interrogation window sizes for each pass (e.g., (64, 32, 16))
- `overlap`: Tuple of overlap values for each pass (e.g., (32, 16, 8))
- `num_iterations`: Number of PIV passes
- `correlation_method`: Method for correlation (`circular` or `linear`)
- `deformation_method`: Method for window deformation (`symmetric` or `second image`)
- `validation_first_pass`: Whether to validate the first pass
- `replace_vectors`: Whether to replace outliers
- Validation thresholds: `min_max_u_disp`, `min_max_v_disp`, `std_threshold`, `median_threshold`, etc.

Example Usage

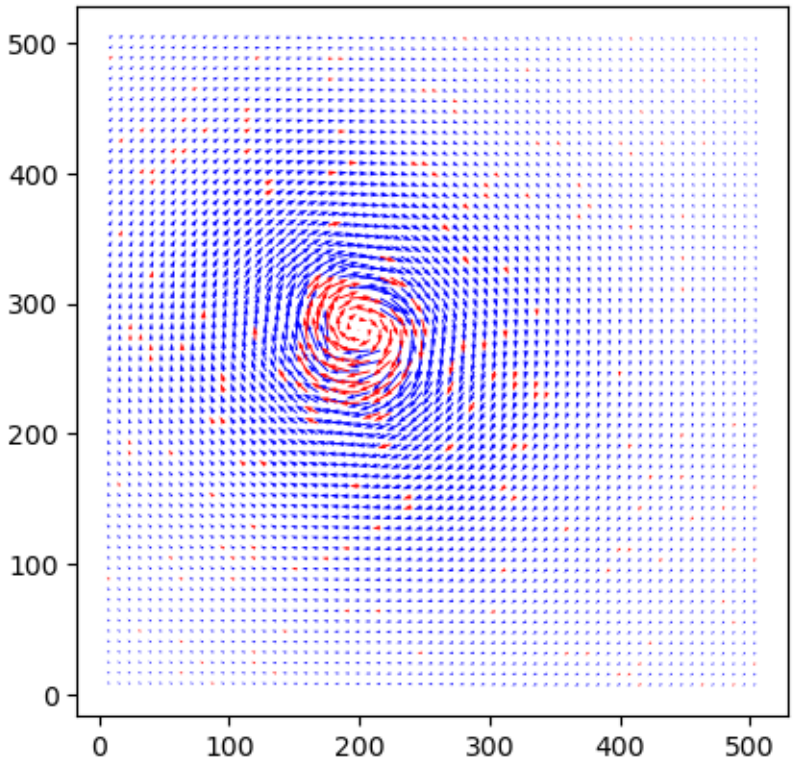
Using advanced PIV techniques in OpenPIV involves setting up `PIVSettings` and calling the `windef.piv()` function:

```
# Import necessary modules
from openpiv import windef
import pathlib

image_path = pathlib.Path(r'./images/')

file_list = []
for path in sorted(image_path.rglob('*.tif')):
    print(f'{path.name}')
    file_list.append(path.name)

# Create settings object
settings = windef.PIVSettings()

# Data related settings
settings.filepath_images = image_path #'./images/' # Folder with the images to process
settings.save_path = './results/' # Folder for the outputs
# Root name of the output Folder (if any) for Result Files
settings.save_folder_suffix = 'test'
# Format and Image Sequence (see below for more options)
#settings.frame_pattern_a = 'exp1_001_b.bmp'
#settings.frame_pattern_b = 'exp1_001_c.bmp'
# or if you have a sequence:
settings.frame_pattern_a = '*.tif'
# settings.frame_pattern_b = '(1+2),(2+3)'
# settings.frame_pattern_b = '(1+3),(2+4)'
settings.frame_pattern_b = '(1+2),(3+4)'

# Format and Image Sequence
#settings.frame_pattern_a = '*.bmp' # file_list[0]

# settings.frame_pattern_b = file_list[1]
#settings.frame_pattern_b = None

# If you want only one pair
#settings.frame_pattern_a = file_list[0]
#settings.frame_pattern_b = file_list[1]


# Region of interest: (xmin,xmax,ymin,ymax) or 'full' for full image
settings.roi = 'full'

# Configure settings for advanced analysis
settings.windowsizes = (64, 32, 16) # it should be a power of 2
settings.overlap = (32, 16, 8) # This is 50% overlap. In general window size/2 is a good choice
```

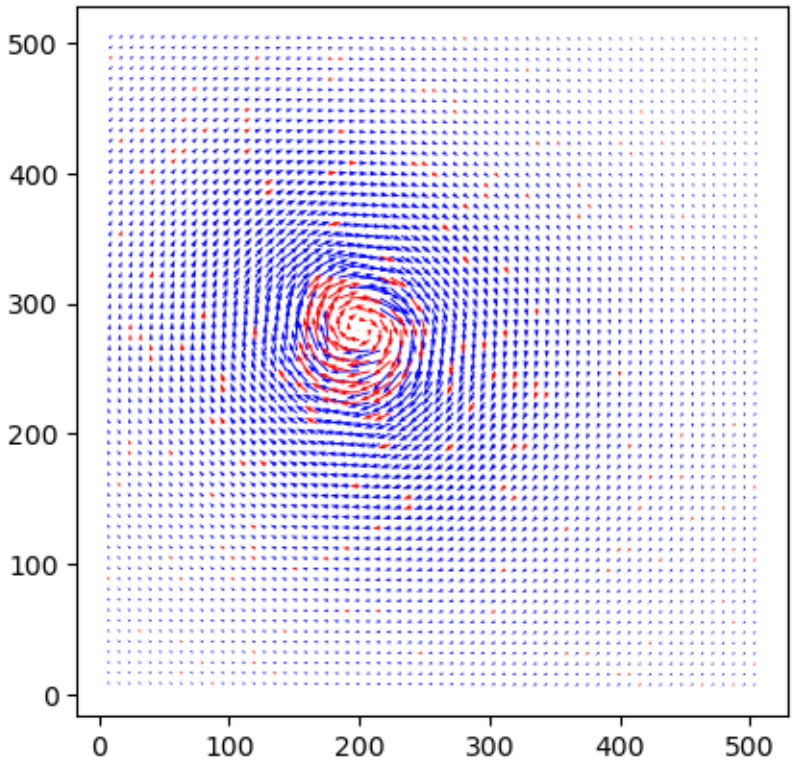
```
settings.num_iterations = 3 # select the number of PIV passes
settings.correlation_method = 'circular' # 'circular' or 'linear'
settings.normalized_correlation = False
settings.subpixel_method = 'gaussian' # 'gaussian','centroid','parabolic'
settings.deformation_method = 'symmetric'
settings.interpolation_order = 3 # order of the image interpolation for the window deformation

# Signal to noise ratio options (only for the last pass)
# It is possible to decide if the S/N should be computed (for the last pass) or not
# If extract_sig2noise==False the values in the signal to noise ratio
# output column are set to NaN
settings.extract_sig2noise = True  # 'True' or 'False' (only for the last pass)
settings.sig2noise_method = 'peak2peak' # 'peak2peak' or 'peak2mean'
# select the width of the masked to masked out pixels next to the main peak
settings.sig2noise_mask = 2

# Set vector validation parameters
settings.validation_first_pass = True # choose if you want to do validation of the first pass
# The validation is done at each iteration based on three filters.
# The first filter is based on the min/max ranges. Observe that these values are defined in
# terms of minimum and maximum displacement in pixel/frames.
settings.min_max_u_disp = (-30, 30)
settings.min_max_v_disp = (-30, 30)
# The second filter is based on the global STD threshold
settings.std_threshold = 7  # threshold of the std validation
# The third filter is the median test (not normalized at the moment)
settings.median_threshold = 3  # threshold of the median validation
# Validation based on the signal to noise ratio'
# Note: only available when extract_sig2noise==True and only for the last
# pass of the interrogation
# Options: True or False
settings.sig2noise_threshold = 1.2 # minmum signal to noise ratio that is need for a valid vect

# Outlier replacement or Smoothing options
# Replacment options for vectors which are masked as invalid by the validation
settings.replace_vectors = True # True or False
settings.smoothn = True #Enables smoothing of the displacement field
settings.smoothn_p = 0.5 # This is a smoothing parameter
# select a method to replace the outliers: 'localmean', 'disk', 'distance'
settings.filter_method = 'localmean'
# maximum iterations performed to replace the outliers
settings.max_filter_iteration = 4
settings.filter_kernel_size = 2  # kernel size for the localmean method

settings.scaling_factor = 1  # scaling factor pixel/meter
```

```
settings.dt = 1  # time between to frames (in seconds)

# Output options
# Select if you want to save the plotted vector field: True or False
settings.save_plot = False
# Choose wether you want to see the vectorfield or not :True or False
settings.show_plot = True
settings.scale_plot = 200  # select a value to scale the quiver plot of the vectorfield

# Run PIV analysis with the given settings
windef.piv(settings)
```

```
B005_1.tif
B005_2.tif
B005_3.tif
B005_4.tif
Saving to results/OpenPIV_results_16_test/field_A0000.txt
```




```
Image Pair 1
B005_1 B005_2
Saving to results/OpenPIV_results_16_test/field_A0001.txt
```




```
Image Pair 2
B005_3 B005_4
```

You can afterwards easily access the computed velocity fields as,

```
# Read x, y, u, v variables from the result file
data = np.loadtxt('results/OpenPIV_results_16_test/field_A0000.txt', skiprows=1)
x = data[:, 0]
y = data[:, 1]
u = data[:, 2]
v = data[:, 3]
```

and estimate/show 2d visualizations:

```
# Create grid for 2D visualization
xi = np.unique(x)
yi = np.unique(y)
X, Y = np.meshgrid(xi, yi)
U = u.reshape(len(yi), len(xi))
V = v.reshape(len(yi), len(xi))
L2 = np.sqrt(U**2 + V**2)
```

```
fig_l2, ax_l2 = plt.subplots(figsize=(6,5))
mesh_l2 = ax_l2.pcolormesh(X, Y, L2, shading='auto')
plt.colorbar(mesh_l2, ax=ax_l2, label='L2 norm')
ax_l2.set_title('L2 norm of velocity')
ax_l2.set_xlabel('x')
ax_l2.set_ylabel('y')
fig_l2.tight_layout()

plt.show()
```

Conclusion

Advanced PIV techniques in OpenPIV provide powerful tools for improving accuracy and spatial resolution in PIV analysis. Window deformation and multi-pass processing address limitations of basic PIV analysis, while comprehensive validation methods ensure reliable vector fields. These techniques are particularly valuable for complex flows with high shear, rotation, or spatial gradients.

Assignement

Now that your are familiar with the advanced PIV techniques, you will be asked to analyze some databases of a particular flow: a Lamb-Oseen vortex flow which is often used for validation due to its well-known analytical solutions. Synthetic images are provided for different particle densities, referred as PPP (Particle Per Pixel).

A `parameters.mat` is also provided. It provides the particle individual locations for each time step (30) in the x and y directions. This file is in a Matlab format but you can access its content as:

```
import scipy.io
mat = scipy.io.loadmat('parameters.mat')
```

Give an evaluation of the velocity vector fields for every test case. Give some insights about the PIV parameters used for the analyses. Evaluate the precision of your estimates with regards to the exact location of the particles as given in `parameters.mat`.

Practice

Measurements

You will conduct experiments with the aim of determining the velocity field of a flow using PIV. The experiments are carried out in a glass aquarium measuring 80 x 35 x 40 cm^3. An adaptable setup inside the aquarium allows for generating a recirculating flow. Its modest volume is well-suited for the space constraints of a practical laboratory exercise. However, you will notice that it has consequences on the established flow that you will analyze.

The flow is generated by 4 Aqua Medic Ecodrift 4.3 pumps. These are propeller pumps commonly used in aquariums. Each pump can produce an adjustable flow rate of up to approximately 4000 l/h, thus creating a controlled recirculating flow in the aquarium.

Image recording

The images are acquired by the FlowSense 2M-165 camera, which connects directly to the computer via a USB 3 port, thus integrating the system directly within the computer without requiring an additional acquisition card. This camera offers a resolution of 1920 $\times$ 1200 pixels (2.3 megapixels), with a maximum acquisition frequency of 165 frames per second. It is therefore well-suited for time-resolved PIV for slow to moderate flows. Its pixel size is 5.9 m, and the quantum efficiency of its sensor is greater than 70%, particularly adapted to the wavelengths of green light from lasers or LEDs. It uses a CMOS (Complementary Metal-Oxide-Semiconductor) sensor, which is widely used in modern PIV systems for its high acquisition rate.

Seeding particles

The seeding particles used are polyamide particles (PSP-50, ref. 9080A5011). These particles are produced by polymerization. They are round, but not perfectly spherical. The size distribution (diameter) of each particle is between 30 and 70 m, with an average of 50 m. Their density is 1.03 g/cm^3, very close to that of water, which therefore limits their sedimentation. Their refractive index is 1.5.

Light source

Illumination is provided by the Fiber-Lite® Mi-LED light generator (Dolan-Jenner). This system delivers white light with a color temperature of 5000 K and represents a modern and economical alternative to conventional 150 W halogen sources.

Assignment

During this practical session, you will get familiarized with a simple experimental setup designed to record particle images of a flow for which the velocity field is to be estimated.

You will be asked to study three particular flows: 1. a free flow (without any obstacle) 2. the flow that develops around/behind a cylinder 3. the flow that develops behind a step

each of which with three flow regimes, i.e. for three different flow rates as given by the pumps. Record sufficiently substantial samples to allow you to subsequently conduct statistical studies.

An in-depth analysis of the obtained images will be expected. Among the points to be discussed, without being exhaustive, should include: the average size of the particle images, their density, and the dynamic range of the gray levels. To do this, you could for example use the `regionprops` library adapted from Matlab.

```
# Import necessary modules
import numpy as np
import matplotlib.pyplot as plt
#matplotlib inline
from openpiv import tools
from skimage.measure import label, regionprops

# Load a sample image
image = tools.imread('your image.tif')
fig, ax = plt.subplots(figsize=(12, 10))
ax.imshow(image, cmap=plt.cm.gray)

# Use a threshold to segment bright particles
threshold = np.percentile(image, 90)  # adjust percentile as needed
binary_img = image > threshold

label_img = label(binary_img, background=0) # adjust background as needed
regions = regionprops(label_img, intensity_image=image)

for props in regions:
    # Use weighted centroid for better accuracy
    y0, x0 = props.weighted_centroid
    ax.plot(x0, y0, marker='x', color='r', markersize=6)

plt.show()
```

You will then work on the collected databases by implementing a method for evaluating velocity fields using OpenPIV. You will detail the method used, emphasizing the choice of parameters for the PIV analysis.

Finally, you will develop an analysis of the obtained velocity fields based on, among other things, considerations discussed in class with Mr. Druault.

References

Adrian, R. J. 1991. "Particle-Imaging Techniques for Experimental Fluid Mechanics." *Annual Review of Fluid Mechanics* 23 (1): 261–304. https://doi.org/10.1146/annurev.fl.23.010191.000245.

Huang, Dabiri, H. 1997. "On Errors of Digital Particle Image Velocimetry." *Measurement Science and Technology* 8 (12): 1427–39. https://doi.org/10.1088/0957-0233/8/12/1427.

Raffel, Willert, M. 2007. *Particle Image Velocimetry: A Practical Guide*. springer.

Rougier, Michael Droettboom, Nicolas P. 2014. "Ten Simple Rules for Better Figures." *PLOS Computational Biology* 10 (9). https://doi.org/10.1371/journal.pcbi.1003833.

Rougier, Nicolas P. 2021. *Scientific Visualization: Python + Matplotlib*. CRC Press.